EEE3027: Multiplier Assignment

Assignment Report

14th of March 2023
URN: 6596386

Department of Electronic Engineering
Faculty of Engineering and Physical Sciences
University of Surrey

Abstract

This report cover the work done on testing an 8-bit multiplier in VHDL and creating as well as testing a 16-bit multiplier as part of EEE3027 Digital Design with VHD. The objectives of this assignment were to learn the skills required to create and analyze VHDL circuits. Further details on the theory and design behind these multipliers including both structural and combinational design are with-in the report. Also included is details of the test benches used and analysis of the simulation data. Finally possible improvements are discussed.

Contents

1   8-Bit Multiplier

For this assignment, the components for this style of multiplier were provided. This includes an 8-bit ripple adder, an 8-bit register, an 8-bit multiplexer, a variable length zero detector, a variable length shift register, a Moore state machine, and the design for the 8-bit multiplier. Note that this final design is structural, with the VHDL code for the multiplier only connecting components together.

1.1   Theory

The core operation of multiplying two numbers, regardless of base, is to multiply the digits, add the product to a total, and then shift the originals numbers[1]. This process can be visualized using "long multiplication" as below ($4_{16} \times D_{16}$):

$$
\begin{array}{ccccccccc}
 & & & & 0 & 1 & 0 & 0 \\
 & & & \times & 1 & 1 & 0 & 1 \\
\hline
 & & & & 0 & 1 & 0 & 0 \\
 & & & 0 & 0 & 0 & 0 & \\
 & & 0 & 1 & 0 & 0 & & \\
 & 0 & 1 & 0 & 0 & & & \\
\hline
 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\tag{1}
$$

For the first step of multiplication binary offers an advantage over other bases as only 4 cases need to be considered: $0 \times 0 = 0$, $1 \times 0 = 0$, $0 \times 1 = 0$, and $1 \times 1 = 1$ (this matches the truth table of an AND gate). To then add this product full adders can be employed. Shifting is the act of multiplying or dividing a number by its base, in the case of binary that is two. In practice this can be moving the connections in opposite directions after each multiply and add. This basic theory will lead us to the combinational design in figure 1.
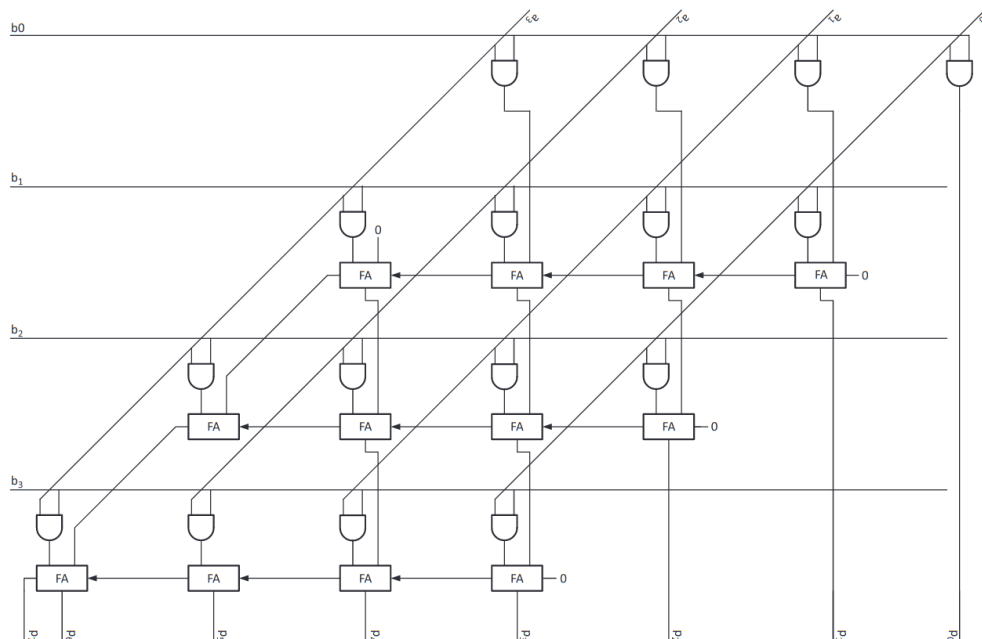

Figure 1: 8-Bit Multiplier using Full Adders [1]

The multiplier achieves all the steps requires as can be seen moving colum to colum. In the first colum multiplication occurs using AND gates. Then all the inputs get shifted over, as can be seen

by the AND gates moving down and the diagonal lines for the A input followed by another multiply and add. This continues, ensuring the carry term also get forwarded appropriately until the output is achieved.

1.1.1 State Machine Multiplier

For this assignment however this combinational design was not used, instead a sequential style of multiplier was provided which is clocked and relies on a Moore state machine to control shifting and adding. Unlike in combinational circuits, which rely only on the current state of the input, sequential designs use feedback to store information about previous inputs[1]. In this case the given circuit is also synchronous meaning that any storage is linked to a clock to ensure all changes happen simultaneously and that races do not occur. Being both sequential and synchronous makes the circuit a Finite-State Machine[1]. Figure 2 shows this alternative multiplier design.

Figure 2: 8-Bit Multiplier using State Machine

F1, the state machine, is at the core of this design and in order to understand the multiplier the operation of the state machine must be understood. One way to visualize a state machine is with a state diagram, such as the one in figure 3 which is based on the design in [2].
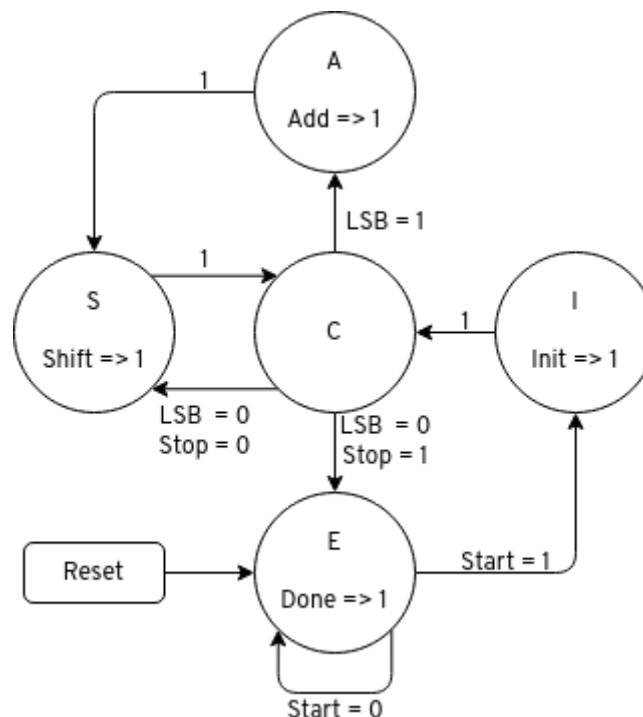
Figure 3: Moore State Machine for Multiplier

Upon reset the state machine will always enter state E, this is also in fact the only asynchronous part of the design as this state change will happen as soon as the reset lines goes high while every other state change will only occur on the rising edge of the clock. In state E the state machine will output '1' on the Done line (any output not labeled as going to '1' is assumed to go to '0') which does not impact any of the other components and is in fact simply and output of the multiplier. As

shown in the diagram while in state E every clock the start signal is checked, as long as it it '0' the state machine remains in state E however once it is high it moves to state I. State I takes the Init line to '1', when tracing this line it can be seen that the following event will occur simultaneously: OR gate REGclr is given an input meaning the gate's output will go to '1' clearing the register of any previous values on the next rising edge. Note that this will also happen when the Reset line goes high as it connects to the other input of the OR gate. The Init line also connects to the LD of both shift registers causing inputs A and B to load into shift registers SR1 and SR2 respectively on the next rising edge. Once in the I state the state machine will always transition to state C. In state C the machine "checks" the value of shift register SR1, in particular if all the bits of SR1's output bus are zero (which is sent by the all zero detect Z1 into the stop line) as well as the value of the least significant bit (LSB). If the LSB bit is '1' the state moves to the A state, otherwise the stop line is checked. When the stop line and LSB bit are '0' the S state is selected and finally if the stop line is '1' and the LSB is '0' the state returns to the E state. When in the A state the Add lines goes to '1', which will make the multiplexer M1 select the output of adder A1 instead of the current value of the register R1. R1 hold the output value meaning that when the Add line is '0' and M1 feed R1 back into R1 nothing will happen, however when R1 is high the output of A1 will be go into R1. A1 adds the output of SR2 together with R1. State A will always transition to state S which takes the Shift line to '1', causing both SR1 and SR2 to shift with SR1 shifting right as DIR is taken low and SR2 shifting left as DIR is taken high. Then state S will transition to state C.

These three states C, A, and S are the core of the multiplying operation and mirror the steps outlined earlier for multiplying binary numbers. Instead of directly multiplying two 4-bit numbers this state machine break the multiplication into multiple 4-bit by 1-bit multiplications. For any N-bit (n) by 1-bit (b) multiplication the following holds: $n \times b = n$ if $b = 1$ or $n \times b = 0$ if $b = 0$. So the way the state machine multiplies two multi-bit binary numbers A and B a shifted copy of B is added for every position A is 1[1]. The shifts form state S will make the LSB the next bit of A and will shift up B at the same time. Once A is all zeros the state machines goes to the end state as no more adding is required.

1.2   Test Bench

The provided test bench code for the 8-bit multiplier was used with some minor changes and corrections. The initially provided code used SRA as a variable name which is invalid VHDL code as SRA is in the reserved list, this however is a simple fix as the variable can renamed throughout the test bench. In order to make the test bench more thorough the range of values investigated was increased to cover very possible 4-bit input (0 to 255) instead of the original limited range, this made finding errors more complicated however due to the large amount of values so another signal was added that keeps a running total of the amount of errors. The error checking is done with an if statement that compares the product of the integer in the for loops to the output out multiplier, if the values do no match an error is logged and the error count in incremented. Keeping track of errors helped during debugging to ensure that the multiplier functioned as expected.

Figure 4: Test Bench of 8-Bit Multiplier

Comprehensive testing ensures that the multiplier will work as expected in every single case

1.3   Timing Analysis

The worst case timing of the 8-bit multiplier can be found by delving into state machine. This is because timing is dictated by the amount of clock cycles, with each full clock cycle being 20ns, and

the state machine will dictated how many clock cycles multiplication takes. Referring to the diagram in figure 3 the shortest journey occurs when the state go from $E \rightarrow I \rightarrow C \rightarrow E$, which occurs anytime A is $A = 0_{16}$ regardless of the value of B. This would take only 60ns to from start input to done output. The longest journey will occur if the multiplier needs to add and shift for every single bit of A, ie when $A = F_{16}$, this is once again independent of the value of B. In this case it would go $E \rightarrow I \rightarrow C \rightarrow A \rightarrow S \rightarrow C \rightarrow A \rightarrow S \rightarrow C \rightarrow A \rightarrow S \rightarrow C \rightarrow A \rightarrow S \rightarrow C \rightarrow E$. This would take 300ns from start to done, which can be confirmed in simulation as seen in figure 5.

Figure 5: Worst Case performance of 8-Bit Multiplier

The average execution time can also be calculated based on the states given a uniform distribution of values of A. While a mathematical proof is possible, for this report all possible values of A and the amount of clock cycles taken were simulated using python code (see appendix) which revealed the average number of clock cycles is 11.125, which would give an execution time, on average, of 222.5 ns. Below is a bar graph, figure 6, of all possible clock cycles times.
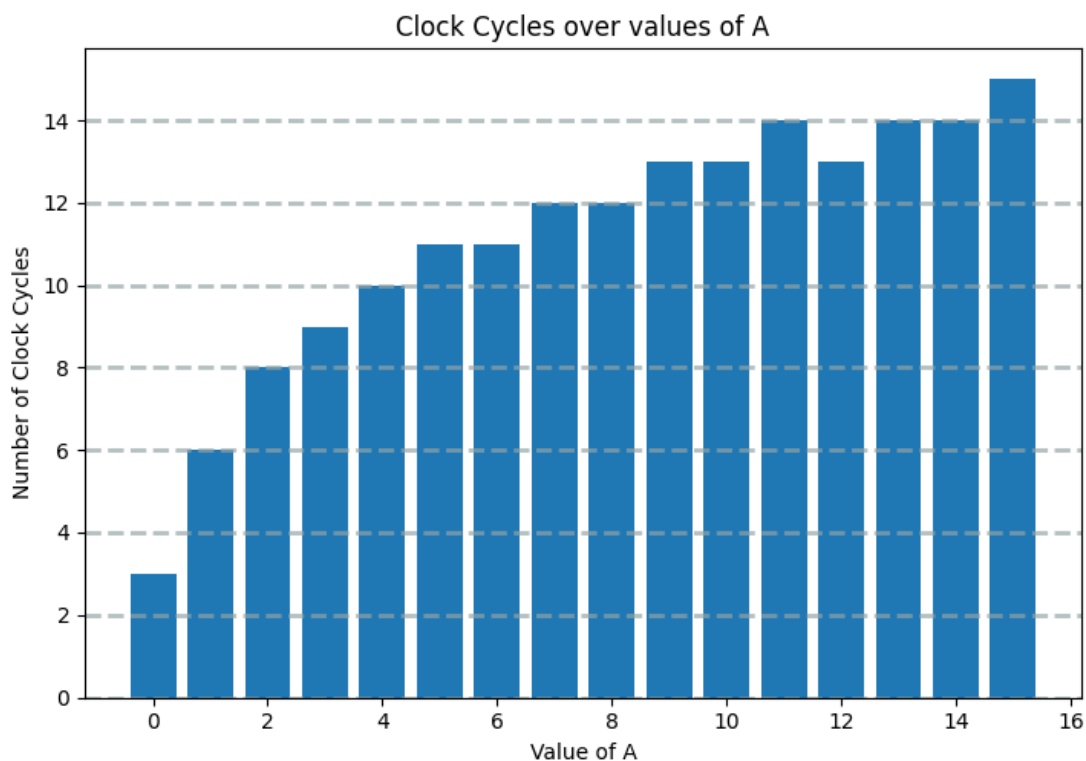

Figure 6: Bar Graph of all Clock Cycles

2 8-Bit Multiplier

8-Bit multipliers can be constructed in various methods, one simple example would be expanding the design seen in figure 1 to take 8-bit inputs and to extend the shift-registers, register, multiplexer and adder in figure 2. These methods however go against some VHDL good practice (as well as against the assignment's instructions), as it is preferred to have smaller components that can be reused throughout a design.

2.1 Theory

For this assignment the 8-bit multiplier was used as a component, along side additional 8-bit adders. This is because 16-bit multiplication can be described as four separate 8-bit multiplication. Take for example the multiplication of 49_16 and $9D_16$ in equation 2 below.

$$
\begin{array}{c}
\begin{array}{cccccccc}
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
\times\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\end{array}
\tag{2}
$$

$$
\begin{array}{cccccccccccccccc}
 & & & & & & & & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
 & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & & & & & & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & & \\
 & & & & & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & & & \\
 & & & & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & & & & \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & \\
 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & & & & & & & \\
\hline
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
$$

As shown by the colors this can be broken down into the following four 4-bit multiplications.

$$
\begin{array}{c}
\begin{array}{ccccc}
 & 1 & 0 & 0 & 1 \\
\times & 1 & 1 & 0 & 1 \\
\hline
 & 1 & 0 & 0 & 1 \\
 0 & 0 & 0 & 0 & \\
 1 & 0 & 0 & 1 & \\
1 & 0 & 0 & 1 & \\
\hline
0\ 1\ 1\ 1\ 0\ 1\ 0\ 1 \\
\end{array}
\end{array}
\tag{3}
\qquad
\begin{array}{c}
\begin{array}{ccccc}
 & 0 & 1 & 0 & 0 \\
\times & 1 & 1 & 0 & 1 \\
\hline
 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 0 & \\
 0 & 1 & 0 & 0 & \\
0 & 1 & 0 & 0 & \\
\hline
0\ 0\ 1\ 1\ 0\ 1\ 0\ 0 \\
\end{array}
\end{array}
\tag{4}
$$

$$
\begin{array}{c}
\begin{array}{ccccc}
 & 1 & 0 & 0 & 1 \\
\times & 1 & 0 & 0 & 1 \\
\hline
 & 1 & 0 & 0 & 1 \\
 0 & 0 & 0 & 0 & \\
 0 & 0 & 0 & 0 & \\
1 & 0 & 0 & 1 & \\
\hline
0\ 1\ 0\ 1\ 0\ 0\ 0\ 1 \\
\end{array}
\end{array}
\tag{5}
\qquad
\begin{array}{c}
\begin{array}{ccccc}
 & 0 & 1 & 0 & 0 \\
\times & 1 & 0 & 0 & 1 \\
\hline
 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 0 & \\
 0 & 0 & 0 & 0 & \\
0 & 1 & 0 & 0 & \\
\hline
0\ 0\ 1\ 0\ 0\ 1\ 0\ 0 \\
\end{array}
\end{array}
\tag{6}
$$

The products of these multipliers can then be added together as seen in equation 7 to get the same result as equation 2. This addition can achieved with three 16-bit adders, which could be constructed out of six of the provided 8-bit adders.

$$
\begin{array}{cccccccccccccccc}
 & & & & & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & & & \\
 & & & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & & & & & \\
 & & & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & & & & & \\
 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & & & & & & & \\
\hline
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\tag{7}
$$

$$
\begin{array}{cccccccc}
 & & & & 0 & 1 & 1 & 1 \\
+ & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\hline
 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}
\tag{8}
$$

$$
\begin{array}{ccccccccc}
 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
+ & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
\hline
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0
\end{array}
\tag{9}
$$

$$
\begin{array}{ccccccccc}
 & & & 0 & 1 & 0 & 0 & 0 \\
+ & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
\hline
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
\tag{10}
$$

However a more efficient design can be created as each of the inputs are "shifted" 8-bit values, this design makes use of only three 8-bit adders and the operation behind it can be seen in equations 8, 9, and 10.

Equation 7 shows how the 3 down to 0 bits of the yellow (3) multiplier's output can be passed directly into the 3 down to 0 bits of the final results. The 7 down to 4 bits of this output however have to be summed with the output of the blue (4), but these cannot be directly added as the upper yellow bits have to be shifted down. This shift can be achieved by wiring the 7 to 4 bits of the yellow to 3 down to 0 inputs of the adders input and keeping the other bits at '0'. The output of this first adder can then be feed into the next adder along with the pink (5) multiplier product to get the next step as seen in equation 9. Once more the 3 down to 0 bits can be passed to the final result (the 7 down to 4 bits) and then the 'shifting' can employed once more to shift the 7 down to 4 bits into the 3 down to 0 bits of the next adders input. Those shifted bits along with the final green multiplier output can be added together in the final third adder as seen in equation 10 to product the last 8 bits of the result.

Doing so introduces a few edge cases however as adders also output a carry line. For adder 1 the carry can be ignored as the largest output of an 8-bit multiplier is $E1_{16}$, and accounting for the shift, the largest values the adder will encounter are $0E_{16} + E1_{16}$ for which the output is EF_{16}. EF_{16} can be conveyed as an 8-bit number meaning the carry will always be '0'. This however is not true for the adder 2 (9) where the largest possible sum is $EF_{16} + E1_{16} = 1D0_{16}$, which means the carry line may go to '1'. Adder 2's carry therefore needs to be propagated, which can be done by connecting it to bit 4 of the adder 3's input, specifically the input with the down shifted number as bits 7 to 4 will always go unused otherwise. Adder 3's carry will also always be '0' as the largest possible values to enter this adder would be $1D_{16} + E1_{16} = FE_{16}$, once more a number that can be conveyed in 8-bits. With all of these steps a 16-bit Multiplier can be constructed out of four 8-bit multipliers and three 8-bit adders.

Figure 7: 16-Bit Multiplier

The final 16-bit multiplier design can be seen in figure 7, note that the inputs and outputs mirror that of the original 8-bit multiplier with a clock, reset, start going in along side two 8-bit values and a single 16-bit output with a done line. All inputs are connected to the inputs of the 8-bit multipliers with-in, with the A and B line being split as describe earlier. The output consist of the result obtained by following equation 7 and the done line waiting for all four multiplier to have output on their done line. An older version of the design also included two 8-bit registers to hold the result just as with the 8-bit design, however these were removed as they served no purpose. While in the 8-bit design

the temporary value of the result is needed in calculation, this is not the case in the 16-bit design and there is no benefit in storing these values currently. If the 16-bit multiplier however made use of different clock cycles for each component having a final register in sync with the main clock would be beneficial.

After creating this design and researching for this report it was discovered that this simple combination of 7 components and few extra logic gates is also known as a Vedic multiplier [3]. The method is based on a traditional ancient indian technique for fast multiplication and if often in used in digital signal processing due to this speed. While this design is faster it is more complex and takes up more space than other types of multipliers.

2.2   Test Bench

Extending the previous test bench from the 8-bit multiplier to this design was simple, as care was taken to ensure the 16-bit multiplier had the exact same inputs and outputs as the 8-bit multiplier. So the original code was extended to input all numbers from 0 to 255 on both ports and the same error count signal was used to aid with early debugging and verification. Initial testing showed errors at the 252, which is binary '11111100', this is also the first instance where the carry bit should be having an influence on the outcome. Using the error signal it was easy to find all the spots with problems and confirm indeed that the carry was an issue, then comparing the VHDL code to the design revealed that indeed the carry had not been connected properly.

Another key note with this test bench as compared to the previous one is that execution time is far longer. While Vivado allowed the default simulation time to be set to 20ms to allow all results to be simulated easily, simulating this much took far longer than the previous simulations for 8-bit. and this shows how comprehensive testing like will eventually cease to be an option once there are too many possible inputs to test and confirm them all.

2.3   Timing Analysis

Timing for this multiplier was simpler, as there are no synchronous elements introduced besides the 8-bit multiplier and the adders will complete well before the end of the clock cycle. Therefore the execution time is as long as the longest 8-bit multiplication and as discussed before the 8-bit multiplier is fully dependant on the A input for its timing with F_{16}. As the new longer A input is split into two, the worst case timing will now occur when either half is F_{16}, ie when $A = XF_{16}$ or $A = XF_{16}$ where X is any hex digits regardless of B. This path can be seen measured in figure 8 and is still 300ns or 15 clock cycles.

Figure 8: 16-Bit Multiplier Worst Case Execution Time

The best case time is still when $A = 00_{16}$ with 60ns. Average timing therefor also stands at 222.5ns.

While the number of clock cycles required in this design cannot be reduced the length of each clock cycle can be. As timings were given for every single component the amount of time needed with-in each clock cycle to complete the longest possible operation can be calculated, this was attempted for

this report and ultimately it was discovered that the clock could be taken all the way from 20ns to 3ns. This reduces the timings significantly however caution is required as these timings may work only in simulation and a more accurate timing report would need to considered the paths between component on the eventual silicon

3   Conclusion and Improvements

This report has covered the theory and testing of both a 8-bit and 16-bit multiplier design, touching on different design philosophies and the timing. These designs are of course not the only option for multipliers, in particular the 8-bit multiplier could make use of the the combinational design mentioned earlier. The combinational design however does not grantee results on a clock cycle as the current design does. Besides the aforementioned increases in clock timings there no other improvements to make on the 8-bit finite state machine design. For the 16-bit multiplier the main improvements consist of improving the 8-bit multiplier with-in, as the design is limited by those. Another improvement would be to replace the ripple adders with look ahead adders, which are more complex and larger but can speed up adding. These speed improvements however are useless unless the goal is also to reduce the clock lengths.

A   Python Code

```python
import math
total_num_clock = 0

values = range(0,16)
num_clocks = []

for value in values:
    num_clock = 2 # From E to I
    bin = f'{value:08b}'
    print(bin)
    for char in bin[::-1]:
        num_clock += 1 # For C
        if value == 0:
            break
        if char == '1':
            num_clock += 1 # For A
        value = math.floor(value/2)
        num_clock += 1 # For S
    print(num_clock)
    total_num_clock += num_clock
    num_clocks.append(num_clock)

average_num_clock = total_num_clock / 16
print(average_num_clock)
```

References

[1] W. J. Dally, R. C. Harting, and T. M. Aamodt, *Digital Design using VHDL A Systems Approach.* Cambridge University Press, 2016.

[2] M. Smith, *Application-specific Integrated Circuits* (Addison-Wesley VLSI systems series). Addison-Wesley, 1997, ISBN: 9780201500226. [Online]. Available: `https://books.google.co.uk/books?id=3hxTAAAAMAAJ`.

[3] K. Keshav Sai Chowdary, K. Mourya, S. Ravi Teja, G. Suresh Babu, and S. Sridevi Sathya Priya, "Design of efficient 16-bit vedic multiplier," in *2021 3rd International Conference on Signal Processing and Communication (ICPSC)*, 2021, pp. 214–218. DOI: `10.1109/ICSPC51351.2021.9451757`.